View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

A Circuit-Based Approach to Efficient Enumeration*†

Antoine Amarilli[1], Pierre Bourhis[2], Louis Jachiet[3], and Stefan Mengel[4]

1 **LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France**
antoine.amarilli@telecom-paristech.fr
2 **CRIStAL, CNRS UMR 9189 & Inria Lille, Lille, France**
pierre.bourhis@univ-lille1.fr
3 **Université Grenoble Alpes, Grenoble, France**
louis.jachiet@inria.fr
4 **CNRS, CRIL UMR 8188, Lens, France**
mengel@cril.fr

Abstract

We study the problem of enumerating the satisfying valuations of a circuit while bounding the *delay*, i.e., the time needed to compute each successive valuation. We focus on the class of *structured d-DNNF circuits* originally introduced in knowledge compilation, a sub-area of artificial intelligence. We propose an algorithm for these circuits that enumerates valuations with linear preprocessing and delay linear in the Hamming weight of each valuation. Moreover, valuations of constant Hamming weight can be enumerated with linear preprocessing and constant delay.

Our results yield a framework for efficient enumeration that applies to all problems whose solutions can be compiled to structured d-DNNFs. In particular, we use it to recapture classical results in database theory, for factorized database representations and for MSO evaluation. This gives an independent proof of constant-delay enumeration for MSO formulae with first-order free variables on bounded-treewidth structures.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases circuits, constant-delay, enumeration, d-DNNFs, MSO

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.111

1 Introduction

When a computational problem has many solutions, computing all of them at once can take too much time. *Enumeration algorithms* are an answer to this challenge, and have been studied in many contexts (see overview in [36]). They generally consist of two phases. First, in a *preprocessing phase*, the input is read and indexed. Second, in an *enumeration phase* that uses the preprocessing result, the solutions are computed one after the other. The goal is to limit the amount of time between each pair of successive solutions, which is called *delay*.

We focus on a well-studied class of efficient enumeration algorithms with very strict requirements: the preprocessing must be *linear* in the input size, and the delay between

* For the full version with proofs, see [3], https://arxiv.org/abs/1702.05589.
† This work was partly funded by the French ANR Aggreg project, by the CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020, by the PEPS JCJC INS2I 2017 CODA, and by the Télécom ParisTech Research Chair on Big Data and Market Insights.


© Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel;
licensed under Creative Commons License CC-BY
44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl;
Article No. 111; pp. 111:1–111:15


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

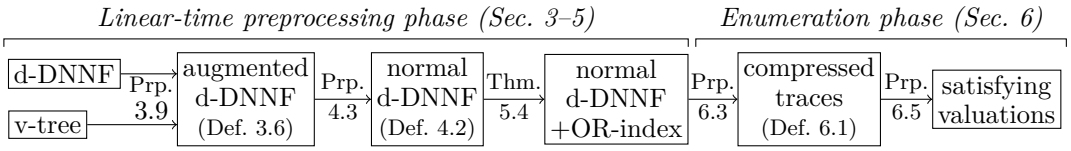

Figure 1 Overview of the proof of Theorem 2.1.

successive solutions must be *constant*. Such algorithms have been studied in particular for database applications, to enumerate query answers (see [19, 6, 20, 7, 8, 25, 24] and the recent survey [33]), or to enumerate the tuples of factorized database representations [29].

One shortcoming of these existing enumeration algorithms is that they are typically shown by building a custom index structure tailored to the problem, and designing ad hoc preprocessing and enumeration algorithms. This makes it hard to generalize them to other problems, or to implement them efficiently. In our opinion, it would be far better if enumeration for multiple problems could be done via one generic representation of the results to enumerate, reusing general algorithms for the preprocessing and enumeration phases.

This paper accordingly proposes a new framework for constant-delay enumeration algorithms, inspired by the field of *knowledge compilation* in artificial intelligence. Knowledge compilation studies how the solutions to computational problems can be *compiled* to generic representations, in particular classes of *Boolean circuits*, on which reasoning tasks can then be solved using general-purpose algorithms. In this paper, we show how this knowledge compilation approach can be implemented for constant-delay enumeration, by compiling to a prominent class of circuits from knowledge compilation called *deterministic decomposable negation normal form* (in short, d-DNNF) [17]. These circuits generalize several forms of branching programs such as OBDDs [18] and were recently shown to be more expressive than Boolean circuits of bounded treewidth [13]. Further, there are many efficient algorithms to compute d-DNNF representations of small width CNF formulae for a wide range of width notions [12], and even software implementations to compute such representations for given Boolean functions [30, 14]. d-DNNFs are also intimately related to state-of-the-art propositional model counters based on exhaustive DPLL [21], to syntactically multi-linear arithmetic circuits [32], and to probabilistic query evaluation in database theory [22].

Our main technical contribution is an efficient algorithm to enumerate the satisfying valuations of a d-DNNF under a standard structuredness assumption, namely, assuming that a so-called *v-tree* is given [31]: this assumption holds in all works cited above. Our first main result (Theorem 2.1) shows that we can enumerate the satisfying valuations of such a circuit with linear preprocessing and delay linear in the Hamming weight of each valuation. Further, our second main result (Theorem 2.2) shows that, if we impose a constant bound on the Hamming weight, we can enumerate the valuations with constant delay. In these results we express valuations succinctly as the set of the variables that they set to true.

To show our results, we consider d-DNNFs under a semantics with implicit negation: variables that are not tested must be set to zero. We call this semantics *zero-suppressed*, like zero-suppressed OBDDs [37]. Our preprocessing rewrites such d-DNNFs to a normal form (Section 4) and pre-computes a multitree reachability index on them (Section 5), allowing us to enumerate efficiently the *traces* of the circuit and the desired valuations (Section 6). To enumerate for d-DNNFs in standard semantics, we show how to rewrite them to zero-suppressed semantics, using the structuredness assumption and a new notion of *range gates* to make the process efficient (Section 3). The overall proof (see Figure 1) is very modular.

Our second contribution is to illustrate how our circuit-based framework and enumeration results can be useful in database theory. As a proof of concept, we present two known results

that we can extend, or recapture with an independent proof. First, we re-prove with our framework that the answers to MSO queries on trees and bounded-treewidth structures can be enumerated with linear preprocessing and delay linear in each assignment, i.e., constant-delay if the free variables are first-order. This was previously shown by Bagan [6] with a custom construction, by Kazana and Segoufin [25] using a powerful result of Colcombet [15], and by Courcelle [16] in a more general setting (but with $O(n \log n)$ preprocessing) using AND/OR-DAGs (that share some similarities with DNNFs). Our proof follows our proposed approach: we compute a circuit representation of the output following the provenance constructions in [4], and simply apply our enumeration result to this circuit. Second, we show how d-DNNFs generalize the deterministic factorized representations of relational instances studied in database theory [29]. We can thus enumerate with linear preprocessing and constant delay for arbitrary deterministic d-representations, which extends the result of [29].

The paper is structured as follows. Section 2 gives the main definitions and results. We then describe the preprocessing phase of our algorithm: we reduce the input circuit to zero-suppressed semantics in Section 3, rewrite it to a normal form in Section 4, and compute the multitree index in Section 5. We then describe the enumeration algorithm in Section 6. We present our two applications in Section 7 and conclude in Section 8. Due to space restrictions, many details and the proofs are found in the complete version [3].

2 Preliminaries and Problem Statement

Circuits. A circuit $C = (G, W, g_0, \mu)$ is a directed acyclic graph (G, W) whose vertices G are called *gates*, whose edges W are called *wires*, which has an *output gate* $g_0 \in G$, and where each gate $g \in G$ has a *type* $\mu(g)$ among $\wedge$ (AND-gate), $\vee$ (OR-gate), $\neg$ (NOT-gate), or var (variable). We represent the circuit with adjacency lists that indicate, for each gate $g \in G$, the gates having a wire to g (called the *inputs* of g), and the gates of which g is an input; the number of such gates is called respectively the *fan-in* and *fan-out* of g. The size $|C|$ of this representation is then $|G| + |W|$. We require that variables have fan-in zero, that NOT-gates have fan-in one, and we will always work on *negation normal form* (NNF) circuits where the input of NOT-gates is always a variable. A circuit without NOT-gates is called *monotone*.

We write C_{var} for the set of variables of C. A *valuation* of C_{var} is a function $\nu : C_{\text{var}} \to \{0, 1\}$. A circuit defines a *Boolean function* on C_{var}, i.e., a function ϕ that maps each valuation ν of C_{var} to $\{0, 1\}$. The image of ν by ϕ is defined by substituting each gate in C_{var} by its value in ν, evaluating the circuit using the standard semantics of Boolean operations, and returning the value of the output gate g_0. Note that AND-gates (resp., OR-gates) with no inputs always evaluate to 1 (resp., to 0) in this process. We call a gate *unsatisfiable* if it evaluates to 0 under all valuations (and *satisfiable* otherwise); we call it *0-valid* if it evaluates to 1 under the valuation which sets all variable gates to 0. We say that ν *satisfies* C if ϕ maps ν to 1 (i.e., g_0 evaluates to 1 under ν), and call ν a *satisfying valuation*.

For enumeration, we represent a valuation ν of C as the set S_ν of variables of C_{var} that it sets to 1, i.e., $\{g \in C_{\text{var}} \mid \nu(g) = 1\}$. We call S_ν an *assignment*, and a *satisfying assignment* if ν is a satisfying valuation. The *Hamming weight* $|\nu|$ of ν is the cardinality of S_ν. Unlike valuations, assignments of constant Hamming weight are of constant size, no matter the size of C_{var}. We write $\{\}$ for the empty assignment, and write $\emptyset$ for an empty set of assignments.

The main class of circuits that we will study are *d-DNNFs* [17], of which we now recall the definition. We say that an AND-gate g of a circuit C is *decomposable* if there is no pair $g_1 \neq g_2$ of input gates to g such that some variable $g' \in C_{\text{var}}$ has a directed path both to g_1 and to g_2: intuitively, a decomposable AND-gate is a conjunction of inputs on disjoint sets of

variables. We say that an OR-gate g of C is *deterministic* if there is no pair $g_1 \neq g_2$ of input gates of g and valuation ν of C such that g_1 and g_2 both evaluate to 1 under ν: intuitively, a deterministic OR-gate is a disjunction of mutually exclusive inputs. A circuit C is a *d-DNNF* if all its AND-gates are decomposable, and all its OR-gates are deterministic.

We further study the subclass of d-DNNFs called *structured d-DNNFs*, i.e., those having a *v-tree* [31]. A *v-tree* on a set S of variables is a rooted unranked ordered tree T whose set of leaves is exactly S. We write $<_T$ for the order on T in which the nodes are visited in a pre-order traversal. For a circuit C, we say that a v-tree T on the set C_{var} is a *v-tree* of C if there is a mapping λ from the gates of C to the nodes of T such that: (i) λ maps the variables of C to themselves; (ii) for each wire (g, g') of C, the node $\lambda(g)$ is a descendant of $\lambda(g')$ in T; and (iii) for each AND-gate g of C with inputs $g_1, \ldots, g_n$ (in this order), the nodes $\lambda(g_1), \ldots, \lambda(g_n)$ are descendants of $\lambda(g)$, none of them is a descendant of another, and we have $\lambda(g_1) <_T \cdots <_T \lambda(g_n)$. Note that having a v-tree implies (by iii) that all AND-gates are decomposable. A *structured d-DNNF* is a d-DNNF C given with a v-tree T of C.

Enumeration. As usual for efficient enumeration algorithms [33], we work in the RAM model with uniform cost measure (see, e.g., [2]), where pointers, numbers, labels for vertices and edges, etc., have constant size; thus an assignment has size linear in its Hamming weight.

An *enumeration algorithm with linear-time preprocessing* computes a set of results $\mathcal{S}(\mathcal{I})$ from an input instance $\mathcal{I}$. It consists of two parts. First, the *preprocessing phase* takes as input an instance $\mathcal{I}$ and produces in *linear time* an indexed instance $\mathcal{I}'$ and an initial *state*. Second, the *enumeration phase* repeatedly calls an algorithm $\mathcal{A}$. Each call to $\mathcal{A}$ takes as input the indexed instance $\mathcal{I}'$ and the current state, and returns a result and a new state: a special state value indicates that the enumeration is over so $\mathcal{A}$ should not be called again. The results produced by the calls to $\mathcal{A}$ must be exactly the elements of $\mathcal{S}(\mathcal{I})$, with no duplicates.

We say that the enumeration algorithm has *linear delay* if the time to produce each new output element $\mathcal{E}$ is linear in the size of $\mathcal{E}$ (and independent of the input instance $\mathcal{I}$). In particular, when the output elements have constant size, each element must be produced with constant delay, which we call *constant-delay enumeration*. The *memory usage* of an enumeration algorithm is the maximum number of memory cells used during the enumeration phase (not counting the indexed instance $\mathcal{I}'$, which resides in read-only memory), expressed as a function of the input instance size $|\mathcal{I}|$ and of the size $|\mathcal{O}|$ of the largest output (as in [6]). Note that constant delay does not imply a bound on memory usage, because the state can become large even if we only add a constant quantity of information at each step.

Main results. Our main theorem on circuit enumeration is the following:

▶ **Theorem 2.1.** *Given a structured d-DNNF C with a v-tree T, we can enumerate its satisfying assignments with linear-time preprocessing, linear delay, and memory usage $O(|\mathcal{O}| \cdot \log |C|)$, where $|\mathcal{O}|$ is the Hamming weight of the largest assignment.*

If we fix a maximal Hamming weight $k \in \mathbb{N}$, we can show constant-delay enumeration:

▶ **Theorem 2.2.** *For any $k \in \mathbb{N}$, given a structured d-DNNF C with a v-tree T, we can enumerate its satisfying assignments of Hamming weight $\leq k$ with preprocessing in time $O(|T| + k^2 \cdot |C|)$, delay in $O(k)$, and memory in $O(k \cdot \log |C|)$, i.e., linear-time preprocessing and constant delay for fixed k.*

In both results, remember that $|C|$ is the number of gates *and wires* of C. We prove our two results in Sections 3–6: the first three sections present the three steps of the linear-time

preprocessing algorithm, and the last one presents the enumeration algorithm. We then use the results for database applications in Section 7, in particular re-proving constant-delay enumeration for MSO queries with free first-order variables on bounded-treewidth structures.

The memory bound in our results is not constant and depends logarithmically on the input. While we think that this is reasonable, we also show constant-memory enumeration for some restricted circuit classes: the details are deferred to [3] for lack of space.

3 Reducing to Zero-Suppressed Semantics

We start our linear preprocessing by rewriting the input circuit to an alternative *zero-suppressed* semantics where negation is coded implicitly. For this rewriting, we will use the structuredness assumption on the circuit, in a weaker form called having a *compatible order*: this is the first thing that we present. We will also extend slightly our circuit formalism, to concisely represent sets of inputs with *range gates* that use this order: we present this second. Last, we present the alternative semantics, and give our translation result (Proposition 3.9).

Compatible orders. Our structuredness requirement is to have a *compatible order*:

▶ **Definition 3.1.** An *order* for a circuit C is a total order $<$ on C_{var}. For two variables $g_1, g_2 \in C_{\mathrm{var}}$, the *interval* $[g_1, g_2]$ consists of the variables g which are between g_1 and g_2 for $<$, i.e., $g_1 \leq g \leq g_2$ or $g_2 \leq g \leq g_1$. The *interval* of a gate g is then $[\min(g), \max(g)]$, where $\min(g)$ denotes the smallest gate according to $<$ that has a directed path to g, and $\max(g)$ is defined analogously. In particular, the interval of any $g \in C_{\mathrm{var}}$ is $[g, g] = \{g\}$.

We say that the order $<$ is *compatible* with C if, for every AND-gate g with inputs $g_1, \ldots, g_n$ (in this order), for all $1 \leq i < j \leq n$, we have $\max(g_i) < \min(g_j)$; in particular, the intervals of $g_1, \ldots, g_n$ are pairwise disjoint.

Note that, if a circuit C has compatible order $<$, every AND-gate g is decomposable: if some $g' \in C_{\mathrm{var}}$ had a directed path to two inputs of g then their intervals would intersect.

Observe further that, given a structured d-DNNF C with a v-tree T, we can easily compute a compatible order $<$ for C in linear time in T. Indeed, let $<$ be the restriction to C_{var} of the order $<_T$ on T given by pre-order traversal. Considering any suitable mapping λ from C to T, for any gate g, we know that $\min(g)$ is no less than the first leaf of T in $<$ reachable from $\lambda(g)$, and that $\max(g)$ is no greater than the last leaf reachable from $\lambda(g)$. The intervals of the inputs $g_1, \ldots, g_n$ to an AND-gate are then pairwise disjoint, because they are included in the sets of reachable leaves from the nodes $\lambda(g_1), \ldots, \lambda(g_n)$ in the v-tree, and none of these nodes is a descendant of another, so they cannot share any descendant leaf. Hence, if we know a v-tree T for C then we know an order $<$ for C.

Augmented circuits. We use compatible orders to define circuits with a new type of gates:

▶ **Definition 3.2.** For $k \in \mathbb{N}$, we define a *k-augmented circuit* C as a circuit with a compatible order $<$ and with k additional types of gates, called *range gates*: there are the *$=i$-range gates* for $0 \leq i < k$, and the *$\geq k$-range gates*. These gates must have exactly two inputs, which must be variables of C (they are not necessarily different, so we allow multi-edges in circuits for this purpose). We talk of *augmented circuits* when the value of k does not matter.

When evaluating a k-augmented circuit under a valuation ν, each $=i$-range gate g (resp., $\geq k$-range gate g) with inputs g_1 and g_2 evaluates to 1 if there are exactly i gates (resp., at least k gates) in $[g_1, g_2]$ set to 1 by ν; note that g may be unsatisfiable if $|[g_1, g_2]|$ is too small.

Range gates are related to the threshold gates studied in circuit complexity (see e.g. [11]), but we only apply them directly to variables. We can of course emulate range gates with standard gates, e.g., ≥ 0-gates always evaluate to 1, and a ≥ 1-range gate on g_1 and g_2 can be expressed as an OR-gate g having the interval $[g_1, g_2]$ as its set of inputs. However, the point of range gates is that we can now write this in constant space, thanks to $<$. This will be important to rewrite circuits in linear time to our alternative semantics.

Zero-suppressed semantics. We are now ready to introduce our alternative semantics for augmented circuits. We will do so only on *monotone* augmented circuits, i.e., without NOT-gates, because negation will be coded implicitly. We use the notion of *traces*:

▶ **Definition 3.3.** An *upward tree* T of a monotone augmented circuit $C = (G, W, \mu, g_0)$ is a subgraph (G', W') of C, with $G' \subseteq G$ and $W' \subseteq W$, which is a rooted tree up to reversing the direction of the wires. For all $(g', g) \in W'$, we call $g' \in G'$ a *child* of $g \in G'$ in T, and call g the *parent* of g' in T; note that g' is an input of g in C. A gate $g \in G'$ in T is an *internal gate* of T if it has a child in T, and a *leaf* otherwise. T is a *partial trace* if its internal gates are AND-gates and OR-gates and if its gates satisfy the following:

- for every AND-gate g in T, *all* its inputs in C are children of g in T;
- for every OR-gate g in T, *exactly one* of its inputs in C is a child of g in T.

Note that T cannot contain OR-gates with no inputs, and that its leaves consist of range gates, variable gates, and AND-gates with no inputs. We call T a *trace* of C if its root is g_0.

We define traces as trees, not general DAGs, because we cannot reach the same gate in a trace by two different paths (remember that AND-gates in augmented circuits are decomposable). We can see each trace (G', W') of $C = (G, W, \mu, g_0)$ as an augmented circuit (G', W', μ, g_0), up to adding to range gates in the trace their inputs in C, and we then have:

▶ **Observation 3.4.** *A valuation ν of a monotone augmented circuit C satisfies C if and only if ν satisfies a trace of C.*

Observe that we can check if a valuation ν of C satisfies a trace T simply by looking at the value of ν on the leaves of T; the definition of ν outside the intervals of the leaves does not matter. We will change this point to define zero-suppressed semantics, where ν can only satisfy T if it maps to 0 all the other variables. We then call ν a *minimal valuation* of T:

▶ **Definition 3.5.** Let C be a monotone augmented circuit, ν be a valuation of C, and T be a trace or partial trace of C. We call ν a *minimal valuation* of T if:

- For every variable g in T, we have $\nu(g) = 1$;
- For every $\bowtie i$-range gate g in T with inputs g_1 and g_2 in C (where $\bowtie \in \{=, \geq\}$ and $i \in \mathbb{N}$), the number n of variables in $[g_1, g_2]$ that are set to 1 by ν satisfies the constraint $n \bowtie i$;
- All other variables of C_{var} are set to 0 by ν.

Note that this implies that ν satisfies T. We call ν a *minimal valuation* for a gate g of C (resp., for C) if it is a minimal valuation of a partial trace rooted at g (resp., at the output g_0).

Note that C may have two minimal valuations ν_1 and ν_2 whose assignments S_1 and S_2 are such that $S_1 \subsetneq S_2$ (see, e.g., Example 3.7 below). Minimality only imposes that, relatively to a trace T, the valuation sets to 0 all variables that are not tested in T. Minimal valuations allow us to define the *zero-suppressed semantics* of a monotone augmented circuit C: the satisfying valuations of C in this semantics are those that are minimal for some trace.

▶ **Definition 3.6.** A monotone augmented circuit C in *zero-suppressed semantics* captures the (generally non-monotone) Boolean function Φ mapping a valuation ν to 1 iff ν is a minimal valuation for C. We call $S(C)$ the set of satisfying assignments of C in this semantics.

We call C a *d-DNNF in zero-suppressed semantics* if it satisfies the analogue of determinism: there is no OR-gate g with two inputs $g_1 \neq g_2$ and valuation ν of C that is a minimal valuation for both g_1 and g_2. (Decomposability again follows from the compatible order.)

▶ **Example 3.7.** Consider the monotone circuit C whose output gate is an OR-gate with three inputs: x, y, and an AND-gate of y and z. The circuit C captures $x \vee y$ in standard semantics, and it is not a d-DNNF. C has three traces, having one minimal valuation each. In the zero-suppressed semantics, we have $S(C) = \{\{x\}, \{y\}, \{y, z\}\}$, and C captures the Boolean function $(x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y)$. Further, C is a d-DNNF in that semantics.

Zero-suppressed semantics makes enumeration easier, because it expresses negation implicitly in a very concise way. The name is inspired by zero-suppressed OBDDs [37, Chapter 8]: variables that are not tested when following a trace are implicitly set to 0. We can equivalently define the assignments $S(C)$ of C inductively as follows:

▶ **Lemma 3.8.** *Let C be a monotone augmented circuit. Let us define inductively a set of assignments $S(g)$ for each gate g in the following way:*

- *for all $g \in C_{\mathrm{var}}$, we set $S(g) := \{g\}$;*
- *for all $\bowtie i$-range gates g with inputs g_1 and g_2, we set $S(g) := \{t \subseteq [g_1, g_2] \mid |t| \bowtie i\}$;*
- *for all OR-gates g with inputs $g_1, \ldots, g_n$, we set $S(g) := \bigcup_{1 \leq i \leq n} S(g_i)$ (with $S(g) = \emptyset$ if g has no inputs);*
- *for all AND-gates g with inputs $g_1, \ldots, g_n$, we set $S(g) := \{S_1 \cup \cdots \cup S_n \mid (S_1, \ldots, S_n) \in \prod_{1 \leq i \leq n} S(g_i)\}$ (with $S(g) = \{\{\}\}$ if g has no inputs); observe that the unions are always disjoint because C has a compatible order.*

Then, for any gate g, the set $S(g)$ contains exactly the assignments that describe a minimal valuation for g. In particular, for g_0 the output gate of C, the set $S(g_0)$ is exactly $S(C)$.

We can now state our main reduction result for this section: we can rewrite any d-DNNF to an equivalent d-DNNF in zero-suppressed semantics, by introducing ≥ 0-range gates to write explicitly that the variables not tested in a trace are unconstrained:

▶ **Proposition 3.9.** *Given a d-DNNF circuit C and a compatible order $<$, we can compute in linear time a monotone 0-augmented circuit C^* having $<$ as a compatible order, such that C^* is a d-DNNF in zero-suppressed semantics and such that $S(C^*)$ is exactly the set of satisfying assignments of C.*

4 Reducing to Normal Form Circuits

In this section, given Proposition 3.9, we work on a monotone 0-augmented d-DNNF circuit C in zero-suppressed semantics, with a compatible order $<$ to define the semantics of range gates. We present our next two preprocessing steps for the enumeration of the assignments $S(C)$ of C: restricting our attention to valuations of the right Hamming weight (for Theorem 2.2 only), and bringing C to a normal form that makes enumeration easier.

Homogenization. Our input augmented circuit C in zero-suppressed semantics may have satisfying assignments of arbitrary Hamming weight. When proving Theorem 2.1, this is intended, and the construction that we are about to describe is not necessary. However, when proving Theorem 2.2 about enumerating valuations of constant weight, we need to restrict

our attention to such valuations, to ensure constant delay. We do so using the following homogenization result, adapted from the technique of Strassen [34]:

▶ **Proposition 4.1.** *Given $k \in \mathbb{N}$ and a monotone augmented d-DNNF circuit C in zero-suppressed semantics with compatible order $<$, we can construct in time $O(k^2 \cdot |C|)$ a monotone augmented d-DNNF circuit C' in zero-suppressed semantics with compatible order $<$ such that $S(C') = \{t \in S(C) \mid |t| \leq k\}$.*

Proof Sketch. We create $k+2$ copies of each gate g, with each copy capturing the assignments of a specific weight from 0 to k inclusive (or, for the $k+2$-th copy, the assignments with weight $> k$). In particular, for $\geq$0-gates g, for $0 \leq i \leq k$, we use an $=i$-gate for the copy of g capturing weight i. We then re-wire the circuit so that weights are correctly preserved. ◀

Note that this is the only place where our preprocessing depends on k: in particular, for constant k, the construction is linear-time. This result allows us to assume in the sequel that the set of assignments of the circuit in zero-suppressed semantics contains precisely the valuations that we are interested in, i.e., those that have suitable Hamming weight.

Normal form. Now that we have focused on the interesting valuations of our circuit C, we can bring it to our desired normal form:

▶ **Definition 4.2.** A *normal* circuit C is a monotone augmented circuit such that:

- C is *arity-two*, i.e., each gate has fan-in at most two.
- C is $\emptyset$-*pruned*, i.e., no gate g is unsatisfiable (i.e., each gate has some minimal valuation).
- C is $\{\}$-*pruned*, i.e., no gate g is 0-valid (i.e., the valuation that sets all variables to 0 is not a minimal valuation for any gate).
- C is *collapsed*, i.e., it has no AND-gate with fan-in 1.
- C is *discriminative*, i.e., for every OR-gate g with an input that is not an OR-gate (we call g an *exit*), g has fan-in 1, fan-out 1, and the one gate with g as input is an OR-gate.

C is a *normal d-DNNF* if it is additionally a d-DNNF in the zero-suppressed semantics.

The pruned requirements slightly weaken the expressiveness of normal circuits C, because they forbid that $S(C) = \emptyset$ or $\{\} \in S(C)$, which are easy to handle separately. We then have:

▶ **Proposition 4.3.** *Given a monotone augmented d-DNNF circuit C in zero-suppressed semantics with compatible order $<$ and with $S(C) \neq \emptyset$ and $S(C) \neq \{\{\}\}$, we can build in $O(|C|)$ a normal d-DNNF C', with $<$ as a compatible order, such that $S(C') = S(C) \backslash \{\{\}\}$.*

Proof Sketch. We reuse the construction of Proposition 4.1 with $k = 1$ to split the gates so that they are not 0-valid, eliminate bottom-up the unsatisfiable gates, make C arity-two in a straightforward way, collapse all AND-gates with fan-in 1, and make C discriminative by inserting new OR-gates (i.e., the exits) on all wires from non-OR-gates to OR-gates. ◀

This result allows us to assume in the sequel that we are working with normal d-DNNFs.

5 Indexing OR-Components

This section presents the last step of our preprocessing. Remember that we now work with a normal d-DNNF, and we want to enumerate its set of assignments. Intuitively, this last preprocessing will help us to enumerate the choices that can be made at OR-gates. Formally, we will work on the *OR-components* of our circuit:

▶ **Definition 5.1.** The *OR-component* K of an OR-gate g in a normal circuit C is the set of OR-gates that can be reached from g by going only through OR-gates, following wires in either direction. We abuse notation and also see K as a DAG, whose vertices are the gates of K, and whose edges are the wires between them.

Recall from Definition 4.2 that, as C is discriminative, all gates of an OR-component K with no inputs in K must be exits; we call them the *exits* of K. For a gate g in K, the *exits* of g are the gates of K that have a directed path to g in K; intuitively, they are the "possible choices" for a partial trace rooted at g. Our goal is to preprocess each OR-component of C to be able to enumerate efficiently the exits of all OR-gates of C. This enumeration task is tricky, however: exploring K naively when enumerating would take time dependent of C, but materializing a reachability index would take quadratic preprocessing time. Thus, we design an efficient indexing scheme, using the fact that OR-components are *multitrees*:

▶ **Definition 5.2.** A DAG G is a *multitree* if it has no pair $n \neq n'$ of vertices such that there are two different directed paths from n to n'. In particular, forests are multitrees, and so are polytrees (DAGs with no undirected cycles).

▶ **Lemma 5.3.** *For any normal d-DNNF C, each OR-component of C is a multitree.*

We can then prepare the enumeration of exits of gates in OR-components, by designing an efficient and generic indexing scheme on *multitrees* (see [3]). We deduce:

▶ **Theorem 5.4.** *Given a normal d-DNNF C, we can compute in $O(|C|)$ a structure called* OR-index *allowing us to do the following: given an OR-gate g of C, enumerate the exits of g in its OR-component K, with constant delay and memory usage $O(\log |K|)$.*

6 Enumerating Assignments

We have described in the previous sections our linear-time preprocessing on the input circuit: this produces a normal d-DNNF C together with an OR-index, and we wish to enumerate its assignments $S(C)$ in zero-suppressed semantics. In this section, we show that we can enumerate the elements of $S(C)$, producing each assignment t with delay $O(|t|)$.

To prove this, we will go back to our definition of zero-suppressed semantics in Section 3, namely, the minimal valuations of the traces of C (recall Definition 3.3). We will proceed in two steps. First, we use our preprocessing and the OR-index to show an efficient enumeration scheme for the traces of C, in a compact representation called *compressed traces*. Second, we show how to enumerate efficiently the minimal valuations of a compressed trace.

Compressed traces. We cannot enumerate traces directly because they can be arbitrarily large (e.g., contain long paths of OR-gates) even for assignments of small weight. We accordingly define *compressed traces* as a variant of traces that collapse such paths:

▶ **Definition 6.1.** An *OR-path* of a monotone augmented circuit $C = (G, W, \mu, g_0)$ is a path from $g \in G$ to $g' \in G$ where all intermediate gates are OR-gates; in particular if $(g, g') \in W$ then there is an OR-path from g to g'. A *compressed upward tree* of C is a pair (G', W') where $G' \subseteq G$ and where $W' \subseteq G' \times G'$ is such that for each $(g, g') \in W'$ there is an OR-path from g to g': we require that T is a rooted tree up to reversing the direction of the edges. T is a *compressed partial trace* if its internal gates are AND-gates and OR-gates such that:

- for every AND-gate g in T, *all* its inputs in C are children of g in T;
- for every exit g in T (it is an OR-gate), its one input in C is a child of g in T;
- for every non-exit OR-gate g in T, *exactly one* of its *exits* g' in C is a child of g in T.

We write $|T| := |G'|$. We call T a *compressed trace* of C if its root is g_0. The *minimal valuations* of a compressed trace are defined like for non-compressed traces (Definition 3.5).

The use of compressed traces is that their size is linear in that of their minimal valuations:

▶ **Lemma 6.2.** *For any compressed trace T of a normal circuit C and minimal valuation ν for T and C, we have $|T| \leq 6 \cdot |\nu|$.*

From a trace T in a normal d-DNNF C, we can clearly define a compressed trace T' with the same leaves, as follows. Whenever T contains an OR-gate g whose parent gate g' in T is not an OR-gate (or when g is the root of T), as g cannot be an exit, we know that there is a OR-path in T from g to an exit g'' of g in its OR-component. We "compress" this OR-path in T' as an edge from g to g''. Conversely, given a compressed trace T', we can fill it to a trace T with the same leaves, by replacing each edge from g to g' by a witnessing OR-path; there is only one way to do so because OR-components are multitrees (Lemma 5.3). Hence, there is a bijection between traces and compressed traces that preserves the set of leaves. As the minimal valuations of traces and compressed traces are defined in the same way from this set, we can simply enumerate compressed traces instead of traces. We can then show:

▶ **Proposition 6.3.** *Given a normal d-DNNF C with its OR-index, we can enumerate its compressed traces, with the delay to produce each compressed trace T being in $O(|T|)$.*

In particular, if all compressed traces have constant size, then the delay is constant.

Proof Sketch. At each AND-gate, we enumerate the lexicographic product of the partial traces of its two children; at each OR-gate, we enumerate its exits using the OR-index. ◀

Enumerating valuations of a compressed trace. We now show how, given a compressed trace T, we can enumerate its minimal valuations (recall Definition 3.5). Restricting our attention to the leaves of T, we can rephrase our problem in the following way:

▶ **Definition 6.4.** The *assignment enumeration problem* for a total order $<$ on gates C_{var} is as follows: given pairwise disjoint intervals $[g_1^-, g_1^+], \ldots, [g_n^-, g_n^+]$, and cardinality constraints $\bowtie_1 i_i, \ldots, \bowtie_n i_n$, where $0 < i_j \leq \left|[g_j^-, g_j^+]\right|$ and $\bowtie_j \in \{=, \geq\}$, enumerate the values of the products $t_1 \times \cdots \times t_n$ for all the assignments of the $t_j \subseteq [g_j^-, g_j^+]$ such that $|t_j| \bowtie_j i_j$ for all j.

Indeed, remember that, as C is $\{\}$-pruned, the leaves of T consist of variables and range gates, and their intervals are pairwise disjoint thanks to decomposability. A $\bowtie i$-gate with inputs g^-, g^+ codes the interval $[g^-, g^+]$ with cardinality constraint $\bowtie i$, and a variable g simply codes $[g, g]$ with constraint $= 1$. Further, thanks to $\{\}$-pruning, we know that no range gate is labeled with $= 0$ or ≥ 0, and thanks to $\emptyset$-pruning, we know that no range gate is labeled with an infeasible cardinality constraint. We claim:

▶ **Proposition 6.5.** *We can enumerate the solutions to the assignment enumeration problem for $<$ on C_{var}, with each solution t being produced with delay linear in its size $|t|$.*

Again, this is constant-delay when all solutions have size bounded by a constant.

Proof Sketch. We enumerate the possible assignments of weights to intervals with constant-delay, to reduce to the case where all cardinality constraints are equalities. We then enumerate the assignments in lexicographic order, using an existing scheme [26, Section 7.2.1.3]. ◀

We have concluded the proof of Theorem 2.1 (see Figure 1) and 2.2. Given our input d-DNNF C and v-tree T rewritten to a compatible order, we rewrite C to an equivalent normal d-DNNF and compute the OR-index. We then enumerate compressed traces, and the valuations for each trace. The proof of Theorem 2.2 is the same except that we additionally use Proposition 4.1 before Proposition 4.3 to restrict to valuations of Hamming weight $\leq k$.

7 Applications

We now present two applications of our main results. Our first application recaptures the well-known enumeration results for MSO queries on trees [6, 23]. The second application describes links to factorized databases and strengthens the enumeration result of [29].

MSO enumeration. Recall that the class of *monadic second-order* formulae (MSO) consists of first-order logical formulae extended with quantification over sets, see e.g. [27]. The *enumeration problem* for a fixed MSO formula $\phi(X_1, \ldots, X_k)$ with free second-order variables, given a structure I, is to enumerate the *answers* of ϕ on I, i.e., the k-tuples $(B_1, \ldots, B_k)$ of subsets of the domain of I such that I satisfies $\phi(B_1, \ldots, B_k)$. We measure the *data complexity* of this task, i.e., its complexity in the input structure, with the query being fixed.

It was shown by Bagan [6] that MSO query enumeration on *trees* and *bounded treewidth structures* can be performed with linear-time preprocessing and delay linear in each MSO assignment; in particular, if the free variables of the formula are first-order, then the delay is constant. This latter result was later re-proven by Kazana and Segoufin [25]. We show how to recapture this theorem from our main results. From the results of Courcelle and standard techniques (see, e.g., [23], Theorem 6.3.1 and Section 6.3.2), we restrict to binary trees.

▶ **Definition 7.1.** Let Γ be a finite alphabet. A Γ-*tree* T is a rooted unordered binary tree where each node $n \in T$ carries a label in Γ. We abuse notation and identify T to its node set. *MSO formulae on* Γ-*trees* are written on the signature consisting of one binary predicate for the edge relation and unary predicates for each label of Γ.

Let $\phi(X_1, \ldots, X_k)$ be an MSO formula on Γ-trees, and let T be a Γ-tree. We will show our enumeration result by building a structured circuit capturing the *assignments* of ϕ on T:

▶ **Definition 7.2.** A *singleton* on $X_1, \ldots, X_k$ and T is an expression of the form $\langle X_i : n \rangle$ with $n \in T$. An *assignment* on $X_1, \ldots, X_k$ and T is a set S of singletons: it defines a k-tuple $(B_1^S, \ldots, B_k^S)$ of subsets of T by setting $B_i^S := \{n \in T \mid \langle X_i : n \rangle \in S\}$ for each i. The *assignments* of ϕ on T are the assignments S such that T satisfies $\phi(B_1^S, \ldots, B_k^S)$.

We will enumerate assignments instead of answers: this makes no difference because we can always rewrite each assignment in linear time to the corresponding answer. We now state the key result: we can efficiently build circuits (with singletons as variable gates) that capture the assignments to MSO queries. (While these circuits are not augmented circuits, they are decomposable, so the definition of zero-suppressed semantics clearly extends.)

▶ **Theorem 7.3.** *For any fixed MSO formula $\phi(X_1, \ldots, X_k)$ on Γ-trees, given a Γ-tree T, we can build in time $O(|T|)$ a monotone d-DNNF circuit C in zero-suppressed semantics whose set $S(C)$ of assignments (as in Definition 3.6) is exactly the set of assignments of ϕ on T.*

Proof Sketch. We simplify ϕ to have a single free variable and limit to assignments on leaves as in [6], and rewrite ϕ to a deterministic tree automaton A using the result of Thatcher and Wright [35], in time independent of T (though the runtime is generally nonelementary in ϕ).

We then compute our circuit as a variant of the *provenance circuits* in our earlier work [4], observing that it is a d-DNNF thanks to the determinism of the automaton as in [5]. This second step is in $O(|A| \cdot |T|)$, so linear in T. A self-contained proof is given in [3]. ◀

Note that the resulting circuit is already in zero-suppressed semantics, and has no range gates. By continuing as in the proof of Theorem 2.1 (for free second-order variables) or of Theorem 2.2 (for free first-order variables), we deduce the MSO enumeration results of [6, 25]. Note that, once we have computed the tree automaton for the query and the circuit representation, our proof of the enumeration result is completely query-agnostic: we simply apply our enumeration construction on the circuit. Our proof also does not depend on the factorization forest decomposition theorem of [15] used by [25]; it consists only of the simple circuit manipulation and indexing that we presented in Sections 4–6. Note that the delay is in $O(k \cdot |T|)$, with no large hidden constants, and $O(k)$ for first-order variables.

A limitation of our approach is that our memory usage bound includes a logarithmic factor in T, whereas [6, 25] show constant-memory enumeration. However, we can show that the circuit computed in Theorem 7.3 satisfies an *upwards-determinism* condition that allow us to replace the indexing scheme of Theorem 5.4 (our memory bottleneck) by a more efficient index. We can thus reprove the constant-memory enumeration of [6, 25]: see [3].

Factorized representations. Our second application is the *factorized representations* of [29], a concise way to represent database relations [1] by "factoring out" common parts. The atomic factorized relations are the empty relation $\emptyset$, the relation $\langle\rangle$ containing only the empty tuple, and singletons $\langle A : a\rangle$ where A is an attribute and a is an element. Larger relations are built using the relational union and Cartesian product operators on sub-relations with compatible schemas. For example, $\langle A_1 : a_1\rangle \times (\langle A_2 : a_2\rangle \cup \langle A_2 : a_2'\rangle)$ is a factorized representation of the relation on attributes A_1, A_2 containing the tuples (a_1, a_2) and (a_1, a_2'). A *d-representation* is a factorized representation given as a DAG, to reuse common sub-expressions. We show that d-representations can be seen as circuits in zero-suppressed semantics:

▶ **Lemma 7.4.** *For any d-representation D, let C be the monotone circuit obtained by replacing $\times$ and $\cup$ by AND and OR, replacing $\emptyset$ and $\langle\rangle$ by AND-gates and OR-gates with no inputs, and keeping singletons as variables. Then all AND-gates of C are decomposable, and $S(C)$ (defined as in Section 3) is exactly the database relation represented by D.*

Hence, our results in Theorem 2.2 can be rephrased in terms of factorized representations:

▶ **Theorem 7.5.** *The tuples of a deterministic d-representation D over a schema $\mathcal{S}$ can be enumerated with linear-time preprocessing, delay $O(|\mathcal{S}|)$, and memory $O(|\mathcal{S}| \log |D|)$.*

Note that the existing enumeration result on factorized representations (Theorem 4.11 of [29]) achieves a constant memory bound, unlike ours, but it applies only to deterministic d-representations that are *normal* (Definition 4.6 of [29]), which we do not assume. Normal d-representations are intuitively pruned and collapsed circuits where *no OR-gate is an input to an OR-gate*: this assumption avoids the need, e.g., for the constructions of Section 5.

8 Conclusion

We have shown how to enumerate satisfying valuations for the structured d-DNNF circuits used in AI, with linear preprocessing and delay linear in each valuation (so constant delay for constant Hamming weight). We applied this to factorized databases, and to MSO query

enumeration [6, 23]. Beyond this, however, our method implies efficient enumeration for all knowledge compilation problems that compile to structured d-DNNFs (see Introduction).

A natural question is to extend our constructions for other tasks, e.g., computing the i-th valuation [6, 9]; managing updates [28]; or enumerating in order of weight, or in lexicographic order: this latter problem is open for MSO [33, Section 6.1] but results are known for factorized representations following an f-tree [10]. Another direction is to strengthen our results to constant-memory enumeration on all d-DNNFs, or generalize them to other classes. We also plan to study practical implementations, because our construction only performs simple and modular transformations on input circuits, with no hidden large constants.

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/pdfs/all.pdf.

2 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

3 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration, 2017. URL: https://arxiv.org/abs/1702.05589, arXiv:1702.05589.

4 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Provenance circuits for trees and treelike instances. In *ICALP*, 2015. URL: https://arxiv.org/abs/1511.08723.

5 Antoine Amarilli, Pierre Bourhis, and Pierre Senellart. Tractable lineages on treelike instances: Limits and extensions. In *PODS*, 2016. URL: https://arxiv.org/abs/1604.02761.

6 Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

7 Guillaume Bagan, Arnaud Durand, Emmanuel Filiot, and Olivier Gauwin. Efficient enumeration for conjunctive queries over X-underbar structures. In *CSL*, 2010. URL: https://hal.inria.fr/hal-00489955.

8 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007. URL: http://www.logique.jussieu.fr/~durand/webperso/papers/BDGlongversion.pdf.

9 Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *ITA*, 42(1), 2008. URL: https://hal-univ-diderot.archives-ouvertes.fr/file/index/docid/221730/filename/bdgo.pdf.

10 Nurzhan Bakibayev, Tomáš Kočiskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *PVLDB*, 2013. URL: https://www.cs.ox.ac.uk/dan.olteanu/papers/bkoz-vldb13-with-response.pdf.

11 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC^1. *JCSS*, 41(3), 1990. URL: http://www.sciencedirect.com/science/article/pii/002200009090022D.

12 Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. On compiling CNFs into structured deterministic DNNFs. In *SAT*, 2015. URL: http://www.dcs.bbk.ac.uk/~florent/publi/cnf-to-ddnnf-upper-bound.pdf.

13 Simone Bova and Stefan Szeider. Circuit treewidth, sentential decision, and query compilation. In *PODS*, 2017. URL: https://arxiv.org/abs/1701.04626.

14 Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *AAAI*, 2013. URL: http://reasoning.cs.ucla.edu/fetch.php?id=128&type=pdf.

15 Thomas Colcombet. A combinatorial theorem for trees. In *ICALP*, 2007. URL: https://www.irif.fr/~colcombe/Publications/ICALP07-colcombet.pdf.

16 Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12), 2009. URL: https://www.labri.fr/perso/courcell/Textes/LinDelayEnum.pdf.

17 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics*, 11(1-2), 2001. doi:10.3166/jancl.11.11-34.

18 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *JAIR*, 17, 2002. URL: https://www.jair.org/media/989/live-989-2063-jair.pdf.

19 Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *TOCL*, 8(4), 2007. URL: https://arxiv.org/abs/cs/0507020.

20 Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In *PODS*, 2014. URL: https://hal.inria.fr/hal-01070898/en.

21 Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, 2005. URL: https://ijcai.org/Proceedings/05/Papers/0876.pdf.

22 Abhay Kumar Jha and Dan Suciu. Knowledge compilation meets database theory: Compiling queries to decision diagrams. *TCS*, 52(3), 2013.

23 Wojciech Kazana. *Query evaluation with constant delay.* PhD thesis, École normale supérieure de Cachan, 2013. URL: https://tel.archives-ouvertes.fr/tel-00919786/document.

24 Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *PODS*. ACM, 2013. URL: https://hal.inria.fr/hal-00908779/en.

25 Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013. URL: https://hal.archives-ouvertes.fr/docs/00/90/70/85/PDF/cdlin-survey.pdf.

26 Donald E. Knuth. *The Art of Computer Programming. Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley, 2005.

27 Leonid Libkin. *Elements of Finite Model Theory.* Springer, 2004.

28 Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *CSL-LICS*, 2014. URL: http://www.theoinf.uni-bayreuth.de/download/lics14-preprint.pdf.

29 Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results. *TODS*, 40(1), 2015. URL: http://www.cs.ox.ac.uk/dan.olteanu/papers/oz-tods15.pdf.

30 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *IJCAI*, 2015. URL: http://reasoning.cs.ucla.edu/fetch.php?id=157&type=pdf.

31 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, 2008. URL: http://aaai.org/Papers/AAAI/2008/AAAI08-082.pdf.

32 Ran Raz, Amir Shpilka, and Amir Yehudayoff. A lower bound for the size of syntactically multilinear arithmetic circuits. *SIAM J. Comput.*, 38(4), 2008. doi:10.1137/070707932.

33 Luc Segoufin. A glimpse on constant delay enumeration (invited talk). In *STACS*, 2014. doi:10.4230/LIPIcs.STACS.2014.13.

34 Volker Strassen. Vermeidung von divisionen. *Journal für die reine und angewandte Mathematik*, 264, 1973. URL: https://eudml.org/doc/151394.

35 James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2(1), 1968.

36 Kunihiro Wasa. Enumeration of enumeration algorithms. *CoRR*, abs/1605.05102, 2016. URL: https://arxiv.org/abs/1605.05102.
37 Ingo Wegener. *Branching programs and binary decision diagrams*. SIAM, 2000.